



Macintosh®

Allegro Common LISP

Version 1.2

Release Notes

Overview

This is the first Apple release of Macintosh Allegro Common Lisp Version 1.2 (software version 1.2.2). This product was previously sold in three separate modules by Coral Software; Allegro CL Version 1.2, Foreign Function Interface, and Stand-Alone Application Generator. Any references to Coral Software are inadvertent and should be disregarded.

The sections in this document follow the same order as the chapter of the Allegro CL User's Guide.

Getting Started

Installation

This section supersedes the section "Installing Allegro CL" on page ii of the main manual. The following procedure installs Allegro CL as well as the Foreign Function Interface and the Stand-Alone Application Generator.

Allegro CL ships on two floppy disks. Neither disk includes a system file. To use Allegro CL, you will need a hard disk with a system file installed. Simply copy the files from both Allegro CL disks to a single folder on your hard disk. If you have older versions of Allegro CL on your disk, you may wish to remove (or archive) them. This will guarantee that when you double-click a document, it will open with the right version of Allegro CL.

The Help File

Allegro CL now includes documentation strings and argument list information for many built in functions (though not all of them). These are accessible through the Fred commands `control-x control-d` (documentation) and `control-x control-a` (argument list). The documentation strings are stored in a separate file, Allegro Help. If this file is not in the `ccl1` logical directory, Allegro will ask you to help locate it the first time you attempt to use documentation strings or argument lists.

The disassembler

Because of size constraints, the disassemble feature has been moved from the main Allegro CL application into a fasl file. This file is in the library folder. It will be loaded automatically (using require) when disassemble is called. The only constraint is that it be kept in a folder directly accessible to Allegro CL.

abort-character

The variable ***abort-character*** has been removed from Allegro CL. This function is now only performed by command-period.

The variable was removed so that control-g could be used by ed-abort-listener-input. This command, when typed in the Listener, aborts the current input and prints a new Listener prompt. The old input is not deleted, so that you can go back to it later.

Allegro CL Menus

There have been several changes in the arrangement of the menubar.

In the **Edit**, the **Change Font** and **Insert Killed String...** menu-items have been removed. These are now available as example files. The **Documents...** menu-item has been removed (because the ccl-doc folder has also been removed).

In the **Eval** menu, menu items have been added for step and trace. The **Step...** menu-item lets the user input a string, which is read and then stepped. The **Trace...** menu-item brings up a dialog box giving options for tracing and untracing functions.

The order of items in the **Tools** menu has been changed. In addition:

- the **Apropos** menu-item brings up a simpler dialog than the one described in the manual. The more complex dialog is given as example source code.

- a **Documentation...** menu-item has been added. This lets the user enter a symbol, and then shows the documentation string of the symbol.

- a **Search Files** menu-item has been added. This functionality was previously available through the inspector. It has been speeded up and made non-modal (so that you can still edit files while the searching is performed). The menu-item brings up a dialog box which lets you search a set of files for a given string. You can specify the file set using wildcard characters, as for the function directory.

Fred

This section describes additions and changes in the use of Fred. Additional documentation on programming Fred is given below.

save-fred-window-positions

[Variable]

If true (the default), the position, size, and selection of a Fred window are restored when the window is re-opened. If nil, this information is not restored.

fred-default-font-spec

[Variable]

The font spec used when new Fred windows are opened. The default font spec is initially ("monaco" 9 :plain).

The Kill Ring and the Scrap

Allegro CL Version 1.2 takes a new approach to combining the clipboard and the kill ring.

The clipboard and the kill ring are now stored as two separate data structures. The clipboard is stored in `*fred-scrap*`, and the kill ring is stored in `*killed-strings*`.

Only the traditional Macintosh commands Cut and Copy move text to clipboard. EMACS commands which kill text do not place the text in the clipboard. Paste also works from the clipboard, ignoring the kill ring.

Any command which kills or copies text (including Cut, Copy, Clear, and various EMACS commands) moves the text to the kill ring. In addition, any text which is killed by side-effect (i.e. by typing or pasting when there is a selection) will also be moved to the kill ring. This mechanism guarantees that important text will not be lost by accidentally deleting it away. The exception to the rule is for simple text (i.e. all whitespace or a single word); when killed by side-effect, these are not copied to the kill ring.

Successive kills with no intervening commands now create a single killed string in the kill ring. That is, the killed strings are concatenated into a single string.

The kill ring is now really a ring (i.e. a circular list). This ring is rotated by the meta-y keystroke.

Multiple Fonts

Fred now has a limited multiple font capability. Runs of characters may use a given font spec, and each window has a current insertion font. The insertion font is the font-spec used when new characters are typed into the window.

The standard window functions `window-font` and `set-window-font` behave in the following way when performed on Fred windows:

`window-font` now returns three values: the font-spec of the insertion font, the font-spec of the character at the insertion point (or the first character in the selection), and a boolean. The boolean will be `t` if the entire selection and the insertion font use the same font-spec; otherwise the boolean will be `nil`.

If there is a selection, `set-window-font` merges the font-specs of the characters in the selection with the given font-spec. There can be multiple font-specs in the selection, and they will all be merged individually. If there is no selection, the current insertion font is merged with the given font-spec, and the result is used as the new insertion font.

Fred does not automatically set the insertion font when you move the insertion point. If the insertion font is Monaco 9, typed characters will appear in Monaco 9, even if you place the insertion point between characters which are in Times bold. The insertion font must always be changed explicitly, by a call to `set-window-font` when there is no selection.

A limitation of Fred window multiple fonts is that all the line heights in the window will be the same. This line height is taken from the largest font used in the window. Once you add a large font, the line height will be retained even if you remove all characters which use this font. To restore the smaller line height you have to close the window and reopen it.

Fred windows retain their font information when they are saved to disk and reopened. An example file shows how to add a set of font/size/style menu-items to the **Edit** menu.

Font information is also retained by the cut/copy/paste operations. This can be disabled by setting the variable `*paste-with-styles*` to `nil` (for example, you may not like to copy things from the Listener, and have them stay bold when they are pasted into another window).

`*paste-with-styles*`

[Variable]

if true (the default), style information is retained when text is copied and pasted. If `nil`, this information is discarded. This variable affects all commands which paste text into a window.

Window Package

There are now three methods for setting a window's package. In addition to the standard mode-line and the function `set-window-package`, a window's package will be set if the first expression in the window is a call to `in-package`. As with a mode-line, this method only works if the expression is present when the window is opened; typing or pasting in an `in-package` statement once the window is open will not set the window's package.

Fred Mini-Buffers

Fred windows now contain mini-buffers for conveying current information to the user. The mini-buffer appears in the bottom of the window, on the left side of the horizontal scroll-bar. (The horizontal scroll-bar is now only one-fourth the width of the window. The rest of the space is used by the mini-buffer.) Many system commands print information in the mini-buffer. In addition, you can set the text of the mini-buffer yourself, as described in the section on Programming Fred, below.

In addition to any other information, mini-buffers display package information for the window. This will be the name of the window's package, if the window has a package, or the name of the value of `*package*`.

Searching and Scrolling

Searching for text with the search dialog or with incremental search (described next) may cause scrolling; this will happen if the text is found in a part of the buffer which isn't currently visible. When scrolling occurs, the found text will be placed a number of lines from the top of the window. The number of lines is determined by the variable `*next-screen-context-lines*`.

Additional Fred Commands

Fred now supports the additional keys found on the Apple Extended Keyboard. These keys include

home	ed-beginning-of-buffer move to the top of the window.
end	ed-end-of-buffer move to the end of the window.
page up	ed-previous-screen scroll up one screen
page down	ed-next-screen scroll down one screen
del	ed-delete-char delete one character to the right of the insertion point.
help	ed-inspect-current-sexp inspects the current expression.

In addition, the function keys are now supported. The function keys for undo, cut, copy, and paste are predefined. Other function keys can be defined by the user, as described below in the section on programming Fred.

The use of the left-arrow and right-arrow keys has been extended. Arrow moves by character, meta-arrow moves by word, and control-arrow moves by expression.

The following Fred commands have been added:

control-s	ed-i-search-forward begins a forward incremental search. Incremental searching is described in a complete section, below.
control-r	ed-i-search-backward begins a reverse incremental search. Incremental searching is described in a complete section, below.
meta-t	ed-transpose-words swaps the words on the left and right sides of the insertion point.
control-meta-t	ed-transpose-sexps swaps the s-expressions on the left and right sides of the insertion point.
control-g	when performed in the listener, aborts the current line of input and prints a new listener prompt. The current input is not deleted, so you can go back to it later.
control-x control-r	ed-read-current-sexp reads the current sexp and prints the result to the listener. This command was previously bound to control-r.
control-x control-space	ed-delete-forward-whitespace deletes whitespace from the insertion point to the next non-whitespace character.
meta-y	ed-yank-pop performs a rotating yank. The first meta-y is typed, it yanks (pastes) the first item in the kill ring. If it is immediately typed again, it removes the old insertion, rotates the kill ring, and yanks (pastes) the next item in the kill ring. This can be done indefinitely, rotating through all the items in the kill ring one at a time. Because the kill ring is circular, you will eventually be back at the beginning. Note that the kill ring remains rotated even after you start doing other things.

Incremental Search

Fred now contains an EMACS-style incremental search. The incremental search is available through the keystrokes control-s (incremental search forward) and control-r (incremental search reverse). `ed-read-current-sexp`—which was previously bound to control-r—is now bound to control-x control-r.

The behavior of incremental search is fairly complicated. However, this complexity was designed to make the incremental search easy to use (as well as powerful). If you have trouble following the description below, just try out incremental searching and you should get the hang of it pretty easily.

Using Incremental Search

When you first type control-s in a Fred window, Fred displays the prompt `i-search` in the window's mini-buffer (or `i-search reverse` for control-r). At this point you can start typing characters to form a search string.

If you first type "f", Fred will immediately search for the next occurrence of "f" following the cursor. If found, it will select this f, scrolling the window, if necessary, to make the text visible.

If you then type "o", Fred will search for "fo", starting with the currently selected f. You can continue typing characters to add to the search string. With each addition, Fred will immediately search for the next occurrence of the string and select the found text. The next occurrence may be a simple extension of the previous found text, or it may be some place further on in the buffer.

Additional Searches

Let's say you type "foo", and Fred finds the string in the buffer, but it is not the right one. You want a later occurrence of the string. Typing control-s a second time will search for the next occurrence of the same string. You can continue typing control-s to search for subsequent occurrences of the string.

When a search fails, Fred will beep, and change the `i-search` prompt to say `Failing i-search`. A search may fail because you type control-s and there are no more occurrences of the string in the buffer, or because you type a character which makes a new string which cannot be found in the buffer. In either case, typing control-s will cause the search to wrap back to the beginning of the buffer.

The behavior of control-r is identical to control-s except that the search is performed backwards. Typing additional control-r's will search backwards in the buffer. When wrapping occurs, the search wraps to the end of the buffer and begins from there.

Backing Up with Backspace

There will be times when you will want to change your mind about what you are searching for. For example, you may type "foot" very quickly, when you meant to type "foom". The backspace key has the effect of undoing the last keystroke. This will delete the character typed from the search string and, if necessary, back up the search to the place it was before the character was typed. Additional backspaces will remove additional characters and back up additional positions.

Backspace can be used to undo the effects of control-s and control-r in addition to the effects of typing characters into the search string. For example, suppose you type "foo" followed by control-s and another control-s. At this point, you will be at the third occurrence of "foo" in the window (assuming there are so many foos). If you then type backspace, incremental search will undo the last keystroke (a control-s), leaving you at the second occurrence of foo. Another backspace will undo another control-s, leaving you at the first foo. Another backspace will undo the next prior keystroke, the letter "o", leaving you at the first occurrence of "fo" in the window.

Terminating Incremental Search

There are a number of ways to terminate an incremental search. These include:

clicking the mouse	performs the mouse action and terminates incremental search.
typing escape	terminates the incremental search, moving the insertion point to the end of the selection (on incremental search) or beginning of the selection (on incremental search reverse).
typing control-g	terminates the search if there were no unfound characters. If there were some unfound characters, these are deleted from the search string and the search continues. If the search is terminated, the cursor is returned to its position before the search began.
typing a Fred command	runs the command and terminates the search.
typing a menu equivalent	runs the menu action and terminates the search.

Doing Another Incremental Search

Fred keeps track of the last string used by incremental search. When you do another incremental search, this string appears in the mini-buffer as a default search string. This makes it easy to search several windows for the same string.

When the default string appears, immediately typing control-s or control-r will search for the string. Typing a backspace will search for the string with one character deleted. Typing anything else will delete the default string and start a search from scratch.

Special Incremental Search Keystrokes

Several keystrokes have special meanings in the context of an incremental search. These are:

control-s	search forward for the next occurrence of the current string.
control-r	search backward for the next occurrence of the current string.
backspace	delete the last character typed and back up.
control-g	if done within a failing search, delete all failed characters. If not done within a failed search, abort incremental search and return cursor to its original position.
control-q	get a character, and insert it quoted into the search string. This is used to search for funny characters (like control-s).
control-w	copy the word following the cursor into the search string.
control-y	copy the line following the cursor into the search string.

Objects

The function `remake-object` gives a new inheritance path to the object it is passed, but it will not propagate this new path to any objects which inherit from the given object. This limitation was necessary for efficiency reasons.

A new proclamation has been added to avoid compiler warning generated by using object-variables within `defobfun`s. Executing:

```
(proclaim '(object-variable (class var1 var2 ...)))
```

will cause all `defobfun`s of *class*, or children of *class* to declare *var1*, *var2*, etc. to be object-variables for the body of the `defobfun`.

New Function

`inherit-from-p` *object parent*

[Function]

Returns true if *object* is an object which inherits from *parent*, otherwise returns nil.

<i>object</i>	any data object.
<i>parent</i>	any object.

Macintosh Basics

The syntax of the turnkey dialogs has been modified for consistency and utility, and one turnkey dialog has been added. The modifications are generally upward compatible with the old syntax. The new syntax is given below.

As previously, if the user clicks in a Cancel button, the modal dialog will perform a throw to the nearest `catch-cancel`.

`message-dialog` *message* &key :ok-text :size :position [Function]

Displays a dialog box holding the string *message*, with a single button. The function returns when the user clicks this button or types Return or Enter.

<i>message</i>	A string to be displayed as the message of the dialog.
:ok-text	The text to be displayed in the button. The default button text is "OK". If this string is too long, it will be clipped.
:position	The position of the dialog. The default position is centered in the upper portion of the screen.
:size	The size of the dialog. The default size is #@ (335 100).

y-or-n-dialog *message* &key :size :position [Function]
 :yes-text :no-text :cancel-text

Puts up a standard Macintosh "Yes", "No", "Cancel" dialog, and returns when the user makes a choice. If :cancel-text is nil, there will be no Cancel button.

<i>message</i>	The message to be displayed in the dialog. This should be a string.
:size	The size of the dialog. The default size is #@ (318 145).
:position	The position of the dialog. The default position is centered at the top of the screen.
:yes-text	The text to be displayed in the yes-button. The default is "Yes". This is the default button of the dialog. If the user clicks in the button or types return or enter, yes-or-no-dialog will return true.
:no-text	The text to be displayed in the no button. The default text is "No". If the user clicks this button, yes-or-no-dialog will return false.
:cancel-text	The text to be displayed in the cancel button. The default text is "Cancel". If the user clicks this button, yes-or-no-dialog will throw to the innermost catch-cancel. If this argument is nil, rather than a string, there will be no Cancel button in the dialog.

get-string-from-user &optional *message initial-string* [Function]
 &key :size :position :ok-text :cancel-text

Prompts the user for a string, which it returns. If the user clicks the cancel button, a throw to :cancel is performed.

<i>message</i>	The message to display in the dialog. The default is "Please enter a string."
<i>initial-string</i>	The string to be displayed as the default string by the dialog. The default for this is the empty string.
:size	The size of the dialog. The default size is #@ (335 100).
:position	The position of the dialog. The default position is centered at the bottom of the screen.
:ok-text	The string to be displayed in the OK button. If the user clicks this button (or types return), get-string-from-user will return the current string.
:cancel-text	The string to be displayed in the Cancel button. If the user clicks this button, get-string-from-user will throw to the innermost catch-cancel.

select-item-from-list *list* &key :window-title [Function]
 :table-print-function
 :selection-type

displays a modal dialog containing the elements of *list* in a table. Returns the item selected by the user, or *nil* if the user click "OK" when no item is selected. If the user clicks the "Cancel" button, a throw to :cancel is performed.

list a list containing items to be displayed in the table. The items are printed using *princ* (i.e. without escape characters).

:window-title the message displayed at the head of dialog.

:table-print-function the print function used by the table in the dialog. The default is *princ*. Using this argument you can specialize the dialog. For example, you could pass a print function that only prints the first element of lists.

:selection-type the type of selection allowed by the table. This should be :single, :contiguous, or :disjoint.

Menus

menu-handle [Menu Variable]

If the menu is installed, this instance variable holds the handle to menu record on the Macintosh heap. If the menu is not installed, *menu-handle* will be bound to *nil*.

The menu-handle can be useful when performing low level operations with the Macintosh ROM. Modifying this value can have dangerous side-effects.

menu-id [Menu Variable]

the numeric id of the menu, used by the Macintosh operating system. Each menu has a unique id.

menu-id-object-alist [Variable]

an association list mapping menu id numbers (used by the Macintosh operating system) to menu objects. For low-level hacking, you may wish to look at this list. However, you should probably never modify it.

menubar-frozen [Variable]

If true, no menubar redrawing will occur. This variable is typically bound to *t* while several menu changes are made. Once the changes are done, a call to the trap *_DrawMenuBar* will draw the new menubar all at once. This mechanism can prevent undo flickering of the menubar.

If you use **menubar-frozen**, it is up to you to call *_DrawMenuBar*. The menubar will not be redrawn automatically.

Hierarchical Menus

Allegro CL now provides full support for hierarchical menus. To create a hierarchical menu, you simply create a menu and treat it as a menu-item. The following example is not necessarily very useful, but it shows how hierarchical menus can be used.

```
? (ask *tools-menu* (menu-deinstall))
NIL
? (ask *eval-menu* (add-menu-items *tools-menu*))
NIL
;now look at the eval menu
? (ask *eval-menu* (remove-menu-items *tools-
menu*))
NIL
? (set-menubar *default-menubar*) ;reset the menubar
```

The Apple Menu

To maintain compatibility with MultiFinder™, the Apple Menu has to be treated differently from other menus. In particular, the Apple Menu can never be removed. Calling `menu-deinstall` or `menu-dispose` on the Apple Menu has been redefined to do nothing. (One implication of this is that if you call `(set-menubar nil)`, the Apple Menu will remain in the menubar.)

If you wish to create an application with its own **About...** menu-item in the Apple menu, you should first remove all the menu-items from the Apple menu and then install your own. You begin with the expression

```
(ask *apple-menu* (apply #'remove-menu-items (menu-items)))
```

Don't worry: the desk accessories won't be removed! Once this is done, you can add your own menu-items to the Apple menu. Any menu-items added will automatically be placed above the desk accessories. Normally, an application will have one **About...** menu-item and one blank line menu-item.

The whole time you work on the Apple menu, it will remain installed.

Window Menu-Items

Allegro CL provides a special class of menu-items for operating on the top window. These are window menu-items. Many menu-items perform their action on the top window. If the top window is the wrong type of window, then the menu-item should be disabled (for example, the **Save As...** menu-item is disabled when the top window is the search dialog). Window menu-items provide an easy way to create menu-items that act on windows. Window menu-items are automatically disabled when the top window is the wrong type.

Every window menu-item should have a symbol for its menu-item-action. This symbol should name a function. When the window menu-item is selected, instead of running its action itself, it ask the window to run the action. In addition, if the symbol is undefined for the top window, then the menu-item will be disabled.

As an example, the **Save** menu-item has the menu-item action `window-save`. If the top window has a definition for `window-save`, then the menu-item is enabled; choosing this menu-item will ask the top window to `window-save`. If the top window doesn't have a definition for `window-save`, then the menu-item is disabled; there will be no way to select the menu-item. To make a new class of windows work with the **Save** menu-item, all you have to do is define `window-save` for the class.

Many of the built in menu-items in Allegro CL are window menu-items, for example, **Save**, **Save As...**, **Revert** and **Print...**

window-menu-item

[Variable]

The window-menu-item class, used for creating window-menu-items.

exist *init-list*

[Menu-item Function]

initializes the window menu-item so that it may be installed in a menu.

init-list

a list of keywords and values for initializing the menu-item. These may be any of the *init-list* options described for menu-items, above.

The `:menu-item-action` specified for a window menu-item is used in a special way. When the menu-item is selected, the top window is asked to run the action. The action should be a symbol naming a function. If the top window has no definition for this function, the window menu-item will be disabled.

Windows

Centering Windows

A new syntax has been added to `exist` for windows, to specify that the window should be centered. This feature is especially useful when creating dialogs.

To center a window, specify the position as the keyword `:centered`, or as a list of the form (*reference offset*).

If the position is `:centered`, the window will be centered vertically or horizontally.

If the position is a list, *reference* should be one of the keywords `:top`, `:left`, `:bottom`, or `:right`, and *offset* should be a number.

If *reference* is `:top`, the top of the window will be *offset* pixels from the top of the screen, and the window will be centered horizontally.

If *reference* is `:bottom`, the bottom of the window will be *offset* pixels from the bottom of the screen, and the window will be centered horizontally.

If *reference* is `:left`, the left side of the window will be *offset* pixels from the left of the screen, and the window will be centered vertically.

If *reference* is `:right`, the right side of the window will be *offset* pixels from the right of the screen, and the window will be centered vertically.

Additional Functions and Functionality

The following functions are implemented and exported in Allegro CL version 1.2.

set-window-font *font-spec*

[Window Function]

In addition to the functionality described on page 6-2, `set-window-font` now merges *font-spec* with the window's previous font-spec.

font-spec a font-specification

For example

```
? (setq bar (oneof *window*))
#<Object #260, "Untitled", a *WINDOW*>
? (ask bar (window-font))
("Geneva" 0 :SRCOR :PLAIN)
? (ask bar (set-window-font '(14 :italic)))
nil
? (ask bar (window-font))
("Geneva" 14 :SRCCOR :ITALIC)
```

exist *init-list*

[Window Function]

In addition to the options described on page 6-1 and 6-2, the `init-list` option `:wptr` is supported. If given, this should be a pointer to a window record on the Macintosh heap. The new window object will be built around this record. This option can be used to integrate new types of windows into the object system (e.g. color windows).

When using this option, the Macintosh window should be created invisibly, and then shown at the end of `exist`.

window-control-click-event-handler *ctlh code* *where mods*

[Window Function]

called by the event system when the user clicks the mouse in a control inside the window.

<i>ctlh</i>	the handle of the control in which the mouse was clicked.
<i>code</i>	the part code of the control in which the mouse was clicked.
<i>where</i>	the mouse position at the time of the click.
<i>mods</i>	a code describing the which modifier keys were held down at the time of the click. This code will be the sum of the following numbers:
	command 256
	shift 512
	caps-lock 1024
	option 2048
	control 4096

window-hardcopy

[Window Function]

may be defined for windows which know how to print their contents to a hardcopy device. If defined, `window-hardcopy` will be called when the window is on top and the user chooses the **Print...** menu-item from the **File** menu.

If this function is not defined for the top window, the **Print...** menu-item will be dimmed.

A sample definition of `window-hardcopy` is given in an example file.

window-zoom-event-handler *code*
[Window Function]

called when the user clicks in the zoom-box of a window. *code* indicates whether the window is zooming to the default position or the user-selected position.

code will be either 7 or 8. If 7, the window is zooming to the user-selected position; if 8, the window is zooming to the default position.

`window-zoom-event-handler` should be defined for windows which need to adjust their contents or redraw whenever they are resized.

`window-zoom-event-handler` should never be called directly. It should only be called by the system, during event-handling.

window-draw-grow-icon [Window Function]

called when a window needs to draw the grow-icon in its lower right corner. You may need to call this function explicitly if you draw over the grow-icon.

When a window is inactive (i.e. not the top window), `window-draw-grow-icon` erases the grow-icon area.

window-grow-rect [Window Variable]
window-drag-rect [Window Variable]

constrain the way in which a window can be resized and moved with the mouse. These variables should always contain rectangle records (though the data in `window-grow-rect` is actually used as two points).

The `toleft` and `bottomright` components of `window-grow-rect` determine the minimum and maximum size that the user can grow the window with the mouse. Note that the user can still zoom the window to other sizes, and the other sizes can be set with the function `set-window-size`.

The `window-drag-rect` rectangle constrains how the window can be moved with the mouse. Whenever the mouse moves outside this rectangle, the gray window outline disappears, indicating that the user is out-of-bounds. (This effect is determined by the position of the mouse, not the potential new position of the window. If the user clicks in the right or left side of the title bar, they will be able to drag the window within a different range.)

Most windows do not have an instance variable for these values. Instead, they use the values inherited from the class `*window*`. If you wish to constrain a particular class or instance, you must first give it its own copy of the variable by asking it to have the variable. If you simply do a `setq` without first doing a `have`, you will be effecting `*window*`'s variables, and all windows will be effected.

Because the rectangles used by `window-drag-rect` and `window-grow-rect` are stored

on the Macintosh heap, they are not garbage collected. Any instance which has one of the rectangles must dispose of it explicitly. This is most conveniently done by calling `dispose-record` from within `window-close`.

Window Event Processing

The following section provides additional information on the redrawing sequence used by windows.

Window Update Events

Whenever a window is created or uncovered, an update event is posted for the window. The next time events are processed, Allegro CL will see the update event and call `window-update-event-handler`. `window-update-event-handler` sets up the `visRgn` of the grafport to be the region that needs updating then calls `window-draw-contents`. The usual version of `window-draw-contents` erases the area within the window's update region (for new windows, this is the entire content area).

Because event processing occurs asynchronously, `window-update-event-handler` may not be called until a little while after a window is created or uncovered (in the default environment, up to 1/3 of a second. See `event-ticks`). This means that drawing performed in the window immediately after the window is created or uncovered may be erased when `window-update-event-handler` is eventually called. To fix this problem, simply call the function `event-dispatch` before drawing into the window. `event-dispatch` forces the processing of any pending events. Note, it is only necessary to call `event-dispatch` when drawing occurs soon after a window is created or uncovered.

Clipping and `window-draw-contents`

When `window-draw-contents` is called, drawing is clipped to the window's update-region. The update-region includes areas which the operating system thinks needs to be redrawn (e.g. the portions of a window which have been covered and then uncovered). To add an area to the update-region, call the trap `_InvalRect` or `_InvalRgn`. To force redrawing of the entire window, make the call

```
(with-port wptr
  (_InvalRect :ptr (rref wptr window.portrect)))
```

Note that the traps `_InvalRect` and `_InvalRgn` cause the Macintosh to post an update event. Because update events are handled asynchronously, these traps may immediately force a call to `window-draw-contents`. If you wish to invalidate several areas before updating, place the calls in the body of a `without-interrupts`. Note also that by the time `window-draw-contents` is called, it is too late to add areas to the update region.

Manual Errata

There is a typo on page 6-3, in the Windows chapter. In the example at the bottom of the page, the call to `exist` should be given the argument `nil`, rather than no argument.

Dialogs

Additional Editable Text Functions

The function `set-current-editable-text` switches the current editable-text item of a dialog. This operation must be done in a single step (because there is always one current editable-text item). The following functions alert an editable-text item when it is about to be exited (and give it a chance to prevent the exit) and when it has just been entered.

`exit-editable-text new-text-item`

[Editable-Text-Dialog-Item Function]

called when an editable-text is about to be exited. At this point, it is still the current editable text, but soon it won't be. This function should return true if it is okay to exit the editable-text item, nil otherwise. If `exit-editable-text` returns nil, the item will remain the current editable-text item. The usual `exit-editable-text` simply returns t.

new-text-item

the item which will be the new current editable text.

`enter-editable-text old-text-item`

[Editable-Text-Dialog-Item Function]

called when an editable-text item has just been made the current editable text.

old-text-item

the item which was previously the current editable text item in the dialog. This may also be nil, the first time an editable text item is added to a dialog.

Additional Table Features

`table-print-function` is a new init-list option for `exist` of `table-dialog-items` that allows the function which prints the items in the cells to be specified. The default value is the function `princ`, which prints the contents without any escape characters. `table-print-function` may be any function with the same calling sequence as `princ` and `princ1`.

`redraw-cell cell`

[Table-Dialog-Item Function]

redraws the display of *cell*. This is useful if you want to update the display of a cell without redrawing the entire table.

cell

the cell to be drawn.

Redrawing the cell involves three operations:

- setting the dialog's clip-rect, so that drawing is restricted to the cell.
- moving the pen to a position three pixels from the bottom and three pixels from the left edge of the cell.
- calling `draw-cell-contents`.

User-defined Dialog-items

New classes of dialog-items can easily be added to the classes of dialog-items which come predefined in Allegro CL.

New classes of dialog-items may be specializations of the specific classes of dialog-items, or they may specialize the abstract class `*user-dialog-item*`. Functions and variables which you may wish to redefine are listed below. In addition, you should probably provide a definition for `dialog-item-draw`, and you may wish to shadow `add-self-to-dialog` and `remove-self-from-dialog`.

An example source code file included with Allegro CL shows how to create icon dialog-items and scroll-bar dialog-items.

`*user-dialog-item*`

[Variable]

Holds the user-dialog-item class, from which user-defined classes of dialog-items should inherit.

`dialog-item-click-event-handler` *where*

[Dialog-Item Function]

called when the mouse is pressed on a dialog-item, *before* the user lets go of the mouse.

where

the location of the mouse-click, in the local window coordinates of the dialog-item's dialog.

The usual `dialog-item-click-event-handler` tracks the mouse and calls `dialog-item-action` if the mouse is released while still over the dialog-item.

This function should be defined for user created classes of dialog-items. It typically highlights the item when the mouse is over the item, unhighlights the item when the mouse is moved away, and call `dialog-item-action` if the button is released while the mouse is within the dialog-item.

`dialog-item-handle`

[Dialog-Item Variable]

dialog-items are often associated with Macintosh data structures, such as controls. If such a data structure is stored in a handle (as they usually are), this handle is stored in the instance variable `dialog-item-handle`. This instance variable will generally be `nil` before the dialog-item is added to a dialog (because the handle is only allocated when the dialog-item is added to the window, i.e. by `add-self-to-dialog`). The instance variable should also be reset to `nil` when the item is removed from the dialog (i.e. by `remove-self-from-dialog`) to avoid having it point to garbage. Static and editable-text dialog-items put a handle to their text in the variable, but only when the item is actually installed in the dialog.

`dialog-item-deactivate-event-handler`

[Dialog-Item Function]

called when the window containing the dialog-item is deactivated (i.e. changed to not be the front window). This function should perform any appropriate behavior. For example, selected text is not highlighted in deactivated windows, and scroll-bars are not shown.

dialog-item-activate-event-handler [Dialog-Item Function]

called when the window containing the dialog-item is activated. This function should perform the inverse of dialog-item-deactivate-event-handler, i.e., it should highlight text, show scroll-bars, etc.

dialog-item-default-size [Dialog-Item Function]

Called by the usual version of add-self-to-dialog. It is called for dialog-items which are not given an explicit size. The usual version of dialog-item-default-size calculates a size based on the font and text of the dialog-item, and the value of the object variable width-correction.

width-correction [Dialog-Item Variable]

used to modify the default width of a dialog-item. A dialog-item or class of dialog-items may be given a binding of this instance variable. The value of width-correction should be an integer. When calculating a default width for an item, this integer is added to the width of the text of the item.

Programming Fred

New Functions and Variables

control-x-comtab [Variable]

holds the comtab accessed by the control-x keystroke.

listener-comtab [Variable]

holds the comtab used by the Listener.

comtabp thing [Function]

returns true if *thing* is a comtab, otherwise returns nil.

get-next-key-event &optional *mini-buffer-text* [Function]

waits for the user to type a keystroke, and returns two values, the message and modifiers field of the event generated by the keystroke. If the user clicks the mouse, nil is returned for both values. (Note that any mouse-clicks will be processed; the user may click in the close box of a window, in which case the window will be closed when get-next-key-event returns.)

If *mini-buffer-text* is given, the text is displayed in the mini-buffer of the current Fred window. If the command is not given in the scope of a Fred window object, the text is not used.

You may wish to decode the values returned by this function with event-keystroke.

fred-special-indent-alist [Variable]

an association-list of symbols and fixnums, indicating that Lisp should provide special indentation when formatting the symbols.

ed-skip-fwd-wsp&comments *buffer start end* [Function]

returns the first position in *buffer* after *start*, skipping over whitespace and comments.

If *start* is not less than *end*, nil is returned.

buffer-word-bounds *buffer position* [Function]

returns two values, giving the starting and ending position of the word surround *position* in *buffer*. The two values will be equal if *position* is not inside a word.

ed-insert-char *character* [Fred Window Function]

inserts *character* into the Fred window at the current cursor position. The display is not automatically updated.

Support for Extended Keyboard Function Keys

The Apple extended keyboard (available for the Macintosh SE and Macintosh II) has a set of function keys. These function keys can now be accessed by Fred. The keystroke names of the function keys are (:function #\1) through (:function #\9), followed by (:function #\a) through (:function #\f).

For example, the following form will bind the F5 function key to to a command for creating a new window:

```
(def-fred-command (:function #\5) ed)
```

The following form will bind the F12 key to a command which prints the date in the top window:

```
(def-fred-command (:function #\c) ;f-12
  (lambda ()
    (multiple-value-bind
      (second minutes hour date month year)
      (get-decoded-time)
      (declare (ignore second minutes hour))
      (format (self) "~a/~a/~a"
                month
                date
                (- year 1900))))))
```

Using the Mini-Buffer

The mini-buffer provides a convenient method for showing information to the user. The information can be the result of a command, a progress indicator, a request for further information, or some combination of all these. All Fred windows display the window's package in the lower-left of the window. This is not considered part of the mini-buffer.

Each Fred window has its own mini-buffer, stored in the instance variable `mini-buffer`. Mini-buffers are output streams, with some additional features.

Note that some of the following functions are Fred Window functions, and some are mini-buffer functions.

set-mini-buffer *format-string* &rest *format-args* [Fred Window Function]

Clears the text of the mini-buffer, prints the *format-string* and *format-args* to the mini-buffer, and then performs a mini-buffer-update to display the mini-buffer.

format-string a format control string, suitable for passing as the second argument to `format`.

format-args a set of *format-arguments*, suitable for passing to `format` along with *format-string*.

mini-buffer-update [Fred Window Function]

Draws the contents of the mini-buffer. This function is normally called whenever the window is updated. You will need to call it explicitly whenever you print to a mini-buffer and wish to show its contents.

stream-reset [Mini-Buffer Function]

Sets the mini-buffer to be empty.

stream-column [Mini-Buffer Function]

Returns the length of the current mini-buffer string.

mini-buffer-string [Mini-Buffer Function]

Returns the string used by the mini-buffer for storing its contents. This will be a vector with a fill-pointer. `stream-tyo` works by pushing characters onto this string. `stream-reset` sets the fill-pointer to zero. `stream-column` returns the value of the fill-pointer.

Using Multiple Fonts

Fred now supports a limited multiple font capability, as outlined above in the section on Using Fred. In addition, the following functions can be used to manipulate fonts and font styles programmatically.

Font spec information is stored with each buffer. Each character in the buffer has a font-spec associated with it. In addition, there is a current insertion font; characters inserted in the buffer will use this font-spec. (Note that the font information is actually stored as a series of ranges in the buffer; there is not a separate font-spec stored for each character.)

Programming with multiple fonts introduces a new data structure, a style vector. A style vector can be mapped over a series of characters in a buffer. The style vector might say, "the first ten characters are in New York 12 bold, then there are twenty characters in Monaco 9 plain, and then 10 characters in Chicago 12 outline." Style vectors are position independent, so that when you apply one, you have to specify a position.

In addition to the functions described below, `buffer-write-file` and `buffer-insert-file` have been modified to preserve and restore font information in a "FRED" resource. Warning: If multiple fonts are being used, and if another text editor is used to edit a file, the font information may become misaligned with the text.

`buffer-set-font-spec` *place font-spec &key :start* [Function]

:end :length

sets the font-spec used in *place*. If a range is specified by *:start*, and *:end* or *:length*, then the given range is changed. If no range is specified, then the current insertion font is changed. In all cases, *font-spec* is merged with the pre-existing font-spec.

<i>place</i>	a buffer or a mark. If a mark, the mark's buffer is used.
<i>:start</i>	the initial position in the buffer to change. This should be a position or a mark. If <i>:start</i> is not given, the current insertion is changed. If <i>:start</i> is given, then either <i>:end</i> or <i>:length</i> must be given.
<i>:end</i>	the final position in the buffer to be affected. This should be a position or a mark.
<i>:length</i>	the number of characters (beginning with <i>:start</i>) to be affected.

`buffer-char-font-spec` *place &optional position* [Function]

returns the font spec of the character at *position* in *place*.

<i>place</i>	a buffer or a mark. If a mark, the mark's buffer is used.
<i>position</i>	a position, mark, or nil. If nil or unspecified, (mark-position <i>place</i>) is used.

`buffer-current-font-spec` *place*
[Function]

returns the current insertion font of *place*. Any text added to the buffer is added in this font.

<i>place</i>	a buffer or a mark. If a mark, the mark's buffer is used.
--------------	---

`buffer-next-font-change` *place &optional position* [Function]

scans the buffer of *place* for the first font change following *position*, and returns the positions of the change. If there are no changes following *position*, nil is returned.

<i>place</i>	a buffer or a mark. If a mark, the mark's buffer is used.
<i>position</i>	a position, mark, or nil. If nil or unspecified, (mark-position <i>place</i>) is used.

buffer-replace-font-spec *place old-spec new-spec* [Function]

modifies the buffer of *place*, so that all occurrences of *old-spec* are replaced with *new-spec*.

<i>place</i>	a buffer or a mark. If a mark, the mark's buffer is used.
<i>old-spec</i>	the font-spec in the window to be replaced.
<i>new-spec</i>	the new font-spec to be used.

This operation actually only involves changing a single entry in a table, and so it is relatively fast.

buffer-get-style *place &key :start :end :length* [Function]

returns a style vector corresponding to the fonts/sizes/styles used in buffer in the specified range.

<i>place</i>	a buffer or a mark. If a mark, the mark's buffer is used.
<i>:start</i>	the start of the range within the buffer. This can be a mark or a number. The default is the buffer-mark.
<i>:end</i>	the end of the range within the buffer. Either <i>:end</i> or <i>:length</i> must be specified, but not both.
<i>:length</i>	the number of character positions, starting with <i>:start</i> , to be included in the style vector. Either <i>:end</i> or <i>:length</i> must be specified, but not both.

buffer-set-style *place style-vector start* [Function]

sets the styles used in *place*, within the range given by *start* and *length*, according to *style-vector*.

<i>place</i>	a buffer or a mark. If a mark, the mark's buffer is used.
<i>style-vector</i>	a style-vector. This should be a vector returned by <i>buffer-get-style</i> .
<i>start</i>	the beginning position in the buffer to insert the styles.

ed-insert-with-style *string style-vector* [Fred Window Function]
&optional *position*

inserts the *string* and applies the *style-vector* over it, at *position* in the window.

<i>string</i>	the string to insert within the buffer. This may also be <i>nil</i> , in which case nothing is inserted.
<i>style-vector</i>	the style-vector to map over the string after it is inserted within the buffer. This may also be <i>nil</i> , in which case the string is inserted with the current insertion font and left unaltered.
<i>position</i>	a position in the window (a mark or a number). The default is the window's cursor position.

The style is not applied if `*paste-with-styles*` is nil. In addition, the Listener shadows this function so that the style is never applied (this helps preserve the Listener history).

Functions which paste should generally work through `ed-insert-with-style`. This will guarantee their behavior is consistent with the rest of the system.

Working With the Kill Ring

ed-delete-with-undo *start end* [Fred Window Function]
 &optional *always-save-p*

deletes the range in the window's buffer specified by *start* and *end*, and returns a cons with the string of the deleted range in the car, and the style-vector of the deleted range in the cdr. In addition, the string/style cons may be pushed onto the kill-ring.

<i>start</i>	the starting position in the window buffer to delete.
<i>end</i>	the ending position in the window to delete.
<i>always-save-p</i>	if true (the default) the deleted text will definitely be added to the kill ring. If nil, the deleted text will only be added to the kill ring if it is non-trivial.

Non-trivial strings are those which contain both word-forming characters and word-delimiting characters. The justification is that text which is explicitly killed should always be saved (therefore *always-save-p* would be true). Text which is killed incidently (e.g. by typing when there is a selection) should only be saved when it is something complicated.

`ed-delete-with-undo` works with the last command mechanism (described in the following section) to concatenate successive kills. If it is called repeatedly, it concatenates the killed text into a single item on the kill ring, rather than creating several items on the kill ring. Deleting with `ed-delete-with-undo` sets the last command to `:kill`.

All the Fred commands which kill text call `ed-delete-with-undo`.

add-to-killed-strings *string-style-cons* [Function]

rotates the kill ring and pushes *string-style-cons* onto the front.

string-style-cons a cons whose car is a string, and whose cdr is a style vector.

rotate-killed-strings *[Function]*

rotates the kill ring. The third item becomes the second, the second item becomes the first, and the first becomes the last. (Remember, the kill ring is a circular list.)

Any empty items are automatically skipped over.

ed-kill-selection *[Fred Window Function]*

deletes the current selection by calling `ed-delete-with-undo`. The killed text is only saved if it is non-trivial (see `ed-delete-with-undo` for a description).

Commands which insert text will general call this function before performing the insertion.

Fred Command Sequence

Page 9-10 of the Allegro CL User's Guide gives a brief description of the sequence of events which occur when Fred processes a keystroke. The following is an expanded version.

The function `window-key-event-handler` for Fred binds the special variable `*current-character*` to the character typed, and binds `*current-keystroke*` to the keystroke code of the keystroke. It then does a look-up on the keystroke, using the function `keystroke-function`. `keystroke-function` will return either a function or a comtab. `window-key-event-handler` then calls `run-fred-command`, passing the function-or-comtab as the argument.

`run-fred-command` checks it's argument.

If it is a comtab, it calls `run-fred-prefix`. `run-fred-prefix` sets the mini-buffer according to `*current-keystroke*`, gets another keystroke using `get-next-key-event`, does a function look-up on this keystroke, and passes the result back to `run-fred-command`.

If it is a function, `run-fred-command` binds `*last-command*` to the object-variable `last-command`, binds `*show-cursor-p*` to `t`, and then sets the object-variable `last-command` to `nil`. It then funcalls the function. When the function returns, if `*show-cursor-p*` is still `t`, `run-fred-command` scrolls to the cursor; otherwise it simply updates the window display.

`last-command`
`*last-command*`

[Fred Window Variable]
[Variable]

an instance variable and special variable used to store information about the last Fred command run in the window. This information is useful for context-sensitive Fred commands. For example, repeatedly calling `ed-yank-pop` (meta-y) inserts successive strings from the kill-ring into a window. For this to work, each call to `ed-yank-pop` needs to know whether the last Fred command was also `ed-yank-pop`.

Whenever a Fred command is run, the special variable `*last-command*` is bound to the current value of the instance variable `last-command`, and `last-command` is set to `nil`. For the duration of the Fred command, functions can check the value of `*last-command*`. A Fred command may also set the value of `last-command`, for the sake of the next Fred command. If `last-command` is not set explicitly, it will be remain `nil`.

`*current-character*`

[Variable]

bound to the character of the current keystroke during the execution of Fred commands. This variable is used by functions (such as `ed-self-insert`) which are bound to several different keys.

current-keystroke

[Variable]

bound to the keystroke code of the current keystroke during the execution of Fred commands.

ed-self-insert

[Fred Window Function]

inserts **current-character** into the window. This function should only be called from within Fred commands (at which time **current-character** is sure to be bound).

run-fred-prefix *comtab*

[Fred Window Function]

gets a keystroke with *get-next-key-event*, looks up the keystroke in *comtab*, and passes the result to *run-fred-command*.

If the keystroke is not explicitly bound in *comtab*, *run-fred-prefix* will pass *ed-beep* to *run-fred-command*.

Debugging

The Toplevel-loop and break-loops

When there is a call to *break*, or when an error is signalled and **break-on-errors** is true, Allegro CL enters a break loop. Breaking does not cause a throw. The break loop is built on top of the previous dynamic environment, so this environment is still accessible.

The stack backtrace (described below) is available when Allegro CL is in a break loop.

You can leave a break loop in two ways. Calling *abort* or typing *command-period* throws to the top level loop, or up one break level. Calling *continue* or typing *command-/* will resume the computation (if the break loop was entered by *break* or *cerror*), or throws to the pending error catch (if the break loop entered by *error*).

Break loops can be entered at any point by calling *break*, typing *command-comma*, or selecting the **break** menu-item.

The Stack Backtrace

The stack backtrace has been improved on several fronts. There is better interaction with the inspector. The names of parameters and local variables can be preserved at the user's discretion. The programmer can easily access and set the values in a stack frame. Finally, information on the program counter and stack frame address is given.

The stack backtrace displays two tables. The upper table displays the pending functions on the stack. Selecting one of these functions will display the function's stack frame in the lower table. In addition, three pieces of information on the frame are given in the space between the tables: the number of values in the frame; the memory address of the frame; and the program counter within the function where execution has been suspended. The memory address is useful for low-level system debugging and for reporting bugs to Coral. The program counter can be used with *disassemble*, to locate the point of a break within a function.

Double clicking a function in the top table will inspect the function object, giving access to edit-definition, documentation, arglist, disassemble, and uncompile.

The stack frame in the lower table will show the names of local variables, if these were retained at compile time. If these were not retained, the parameters are listed as required, optional, key, or rest. The values of these frames can be accessed using the `local` macro, described below. In addition, double clicking a value will inspect the value.

save-local-symbols [Variable]
fasl-save-local-symbols [Variable]

save-local-symbols determines whether local symbols are retained when functions are compiled interactively. If it is true, the symbols are retained. The default is `nil`.

fasl-save-local-symbols provides a default value for the `:save-local-symbols` keyword argument to `compile-file`. If this is true, the local symbols are saved in the compiled file, and they will be available when the file is loaded. The default is `nil`.

Saving local symbols generally increases the size of fasl files by about 15% to 20%.

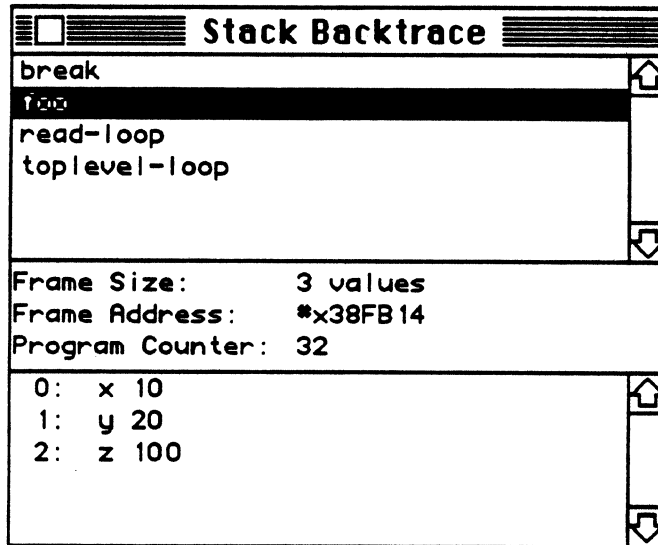
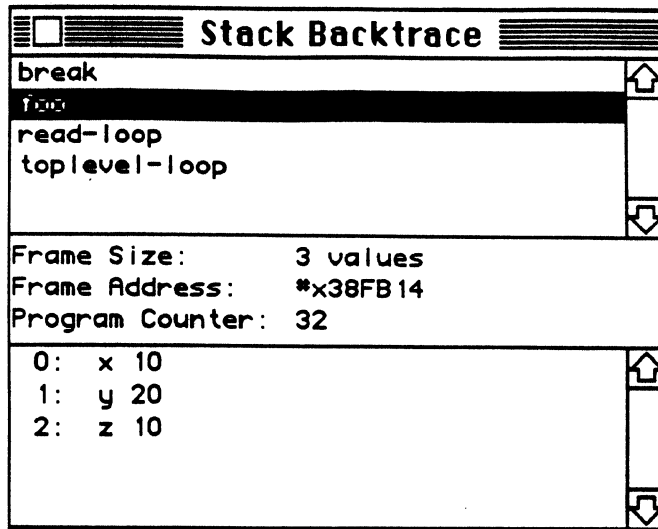
local indicator [Macro]

returns the value in the current stack frame of the slot given by *indicator*. This macro can only be used when a backtrace dialog is visible, and when a frame is selected.

indicator a symbol or number indicating a slot in the stack frame. A symbol can be used if the frame includes local symbol names, and if the symbol is unique in the frame. Otherwise a number giving the position in the frame should be used.

`local` can be used with `setf` to modify a value in a stack frame. This should be done with caution, as the compiler may have made assumptions based on the initial values.

The variable `*d*` (renamed `*debug-value*` in version 1.1) has been removed.



```
? (defun foo (x y)
  (let ((z 10))
    (break)
    (+ x y z)))
foo
? (foo 10 20)
> Break:
> Type Command-/ to continue,
Command-. to abort.
1 > (local x)
10
1 > (local 1)
20
```

```

1 > (setf (local z) 100)
100
1 > (continue)
Continuing...
130

```

Step

As specified by Common Lisp, `step` is equivalent to `eval`, except that it provides interactive features for evaluating forms one sub-form at a time.

Because of this, `step` is performed in the null lexical environment and current dynamic environment. If you insert a call to `step` in the middle of a function, the arguments to the function and other lexical entities will not be visible inside the body of the form being stepped.

Additional Functions and Variables

trace-print-length [Variable]
trace-print-level [Variable]

The values of these variables are used to rebind `*print-length*` and `*print-level*` during trace output. (`*print-length*` and `*print-level*` are described in *Common Lisp: the Language*, on page 372).

print-db &rest forms [Macro]

equivalent to `progn`, except that it prints each *form* and the result of evaluating each *form* as the forms are evaluated. All the printing is sent to `*debug-io*`.

`print-db` is useful for tracking the values of variables, and flow of control inside programs.

Like `progn`, `print-db` returns the value of the last *form*.

System Throws and Error Handling

Future versions of Allegro CL will implement the Common Lisp Error System. The macros outlined below give a simple ability to catch all system throws (including error throws), so that they can be handled programmatically. These features are important for programs which completely take over the top-level loop.

Catching Errors

When the function `error` is called, Allegro CL prints the error message (unless a `catch-error-quietly` is active) and performs one of two actions: If `*break-on-errors*` is false, Lisp throws to the innermost pending error catch. If `*break-on-errors*` is true, Lisp enters a break loop. If the break-loop is exited with `continue`, Lisp will then throw to the innermost pending error catch; if the break-loop is exited by `abort`, a normal abort occurs.

catch-error {form}*

[Macro]

sets up an error catch, and evaluates the *forms*. `catch-error` returns two values. The first value will either be the value of the last *form* (if there was no error), or a list of an error string and arguments (if there was an error). The second value will be `nil` if there was no error, or `t` if there was an error.

The error message will still be printed. In addition, if `*break-on-errors*` is true, the error will still enter a break loop. The throw will only occur when the user leaves the break loop by calling `continue`. In cases where you want to handle the error completely, use `catch-error-quietly`.

catch-error-quietly {form}*

[Macro]

performs the same action as `catch-error`. In addition, the printing of the error message is suppressed, and break-loops will not be entered (i.e. `catch-error-quietly` has the effect of binding `*break-on-errors*` to `nil`).

This macro lets the programmer completely handle error situations.

For example, one shortcoming of the Common Lisp `read` function can be overcome:

```
? (defun safe-read (stream &optional error-val
                    &aux val errorp)
  (multiple-value-bind (val errorp)
    (catch-error-quietly
     (read stream))
    (if errorp
        error-val
        val)))

safe-read
? (setq foo (make-string-input-stream "(incomplete)"))
#<Object #213, a ccl::*string-input-stream*>
? (safe-read foo 'no-good)
no-good
? (setq foo (make-string-input-stream "(complete)"))
#<Object #214, a ccl::*string-input-stream*>
? (safe-read foo 'no-good)
(complete)
```

Catching Aborts

The function `abort` is used for terminating operations and for moving up break levels. `abort` works by throwing to the nearest abort catcher.

catch-abort {form}*

[Macro]

sets up an abort catch, and evaluates the *forms*. `catch-abort` will return the value of the last *form* if there was no abort. If there was an abort, it will return the symbol `:abort`.

Catching Cancels

Modal dialogs—and perhaps other operations—sometimes provide a cancel feature. In previous versions of Allegro CL, this was performed by simply throwing to `:cancel` (or passing `:cancel` to `return-from-modal-dialog`). In version 1.2, the behavior has been encapsulated in the operations `cancel` and `catch-cancel`.

cancel

[Function]

throws to the nearest cancel catch. This function is generally called when the user clicks in the cancel button of a modal dialog.

catch-cancel {form}*

[Macro]

sets up a cancel catch, and evaluates the *forms*. `catch-cancel` will return the value of the last *form* if there was no cancel. If there was a cancel, it will return the symbol `:cancel`.

Pascal Records

Loading Record Definitions

Most of the predefined record types described in the Allegro CL User's Guide are no longer predefined. Instead, they are given as source code in the file `Records.Lisp`, in the Library folder. If you use records, you should load this file, perhaps with the expression `(require 'records)`.

The rectangle record definition is still predefined by Allegro CL.

Efficiency Improvements

The most common record functions have been redefined as macros. This lets them compile into faster code, and it also means that record definitions don't need to be loaded at run-time, only at compile-time.

The following operations are now macros:

```
defrecord  
make-record  
dispose-record  
rlet  
rset  
rref  
copy-record
```

Four other changes have been made for the sake of efficiency:

`dispose-record` now takes an optional second argument. If supplied, this should be a record-type. Supplying the second argument helps the macro expand into faster, more compact code.

Supplying the initial record contents in a call to `rlet` no longer involves a significant performance penalty.

Default records are no longer used by `make-record`. The initial contents of new records is undefined, except for those slots given explicit initial values.

Whenever record-types are specified (besides in `defrecord`), they must be specified as keywords. For example, the form

```
(rlet ((r rect)) . . .)
```

should be replaced with

```
(rlet ((r :rect)) . . .)
```

Extensions to rlet

The storage slots specified to `rlet` can be the names of record field types, as well as record types. So, for example, you can make the call

```
(rlet ((p :point)) . . .)
```

This will allocate enough storage to hold a point (i.e. 4 bytes). When you allocate storage using record field types, you cannot specify the initial contents.

System Interface

Loading Traps

The definitions for Macintosh traps are no longer compiled into Allegro CL. If you want to use the trap definitions, you should load the file `Traps.Lisp` from the Library folder. You may want to selectively comment-out portions of this file, to avoid loading lots of traps you don't need. Loading the entire file uses up about 50K of heap space. The simplest way to ensure the presence of trap definitions is to include the form `(require 'traps)`.

Dereferencing Handles

Handles cannot be dereferenced by simply calling `(%get-ptr my-handle)`. Doing this will sometimes return a value of the wrong type.

The safest way to dereference handles is with the macros `with-dereferenced-handles` and `with-pointers`.

Where speed optimization is need, handles can be dereferenced inline using the following technique. Note that this method works in the current version of Allegro CL, but will probably not work in future versions.

```
(%int-to-ptr (ilogand #xFFFFFFF  
                    (%get-ptr handle)))  
; returns the dereferenced handle
```

Boolean Values

Macintosh traps return boolean values as fixnums. This fixnum will have bit 8 turned on if the value is true.

For example, `(logbitp 8 (_Button :word))` will return true if the mouse-button is down.

When passing boolean values to the Macintosh operating system, use -1 as true, and 0 as false.

Disposing of Pointers

After disposing of Macintosh pointers, it is important to zero all Lisp data structures holding the pointer (e.g. by setting them to `nil`). If this isn't done, Lisp will be maintaining a dead pointer. This can lead to problems if the Macintosh and Lisp heaps resize, because the dead pointers may begin pointing into active Lisp pages.

Stack Blocks

It is now possible to allocate stack blocks whose size is determined at run time, rather than at compile time.

`%vstack-block` (*symbol size*) {*form*}*
[Special Form]

is equivalent to `%stack-block` except that only one *symbol / size* pair may be given, and the *size* can be any form (it doesn't have to be a compile-time constant).

Manual Errata

The function documented as `%inc-pointer` is actually named `%inc-ptr`.

The arguments of the `defpascal` example `my-track-proc` are incorrect. The first line should read:

```
(defpascal my-track-proc ((my-control :ptr) (partCode :word))
```

Files

When `load` is passed a filename with no type, it checks for the existence of the file with no type before checking for files with a `.lisp` or `.fasl` added to the file name. In the past `load` always merged with `.lisp` and `.fasl`, so that it was impossible to load files with no type.

The `:directory` keyword to the function `choose-new-file-dialog` can specify a filename as well as a directory. This filename is used as the default filename when the dialog is displayed.

The variable `*open-file-streams*` is bound to a list of all streams open to files. This variable is useful for closing file streams in an unusual circumstance.

Customizing Streams

Allegro CL implements streams as a simple class of objects. There are only a few things a stream needs to know how to do.

Output streams need to know how to write characters. Functions which output larger amounts of data (such as `print` and `format`) do so by repeatedly writing single characters.

Input streams need to know how to read characters and un-read characters (un-reading allows "peeking ahead"). Functions which input longer series of data do so by repeatedly reading single characters. Input streams also need to know how to tell when they are at end of file.

Characters may be sent to streams as either characters or fixnums representing ASCII values. The following form will guarantee that a fixnum-or-character is converted into a character:

```
(if (fixnump foo) (code-char foo) foo)
```

stream

[Variable]

the class from which all other streams inherit. This is an abstract class. It should not be directly instantiated, but instead used for the creation of new subclasses.

stream supplies definitions for all stream operations. The definitions for the required stream functions (`stream-tyo`, `stream-tyi`, `stream-eofp`, and `stream-untyi`) signal an error. In classes which inherit from ***stream***, any of these functions which will be used should be redefined. The remaining stream operations either perform a reasonable default operation, or else do nothing.

stream-tyo char

[Stream Function]

directs the stream to output the character *char*. For example, if the stream is a file, the character will be written to disk; if it is a window, the character will be displayed on the screen. Output functions such as `print`, `write`, and `pprint` work by repeatedly calling `stream-tyo`.

`stream-tyo` must be defined for all output streams.

char a character or ASCII value.

stream-tyi

[Stream Function]

reads the next character from the stream and returns it. Input functions such as `read` and `read-line` work by making repeated calls to `stream-tyi`.

`stream-tyi` must be defined for all input streams.

stream-untyi char*[Stream Function]*

un-reads *char* from the stream, effectively pushing it back onto the head of the stream. The next call to **stream-tyi** will return *char*. **stream-untyi** cannot be called several times in a row; it can only be called once for each call to **stream-tyi**.

stream-untyi must be defined for all input streams.

char should be the last character read from the stream.

stream-untyi is usually implemented in one of two ways: If the stream contains a pointer to a file, string, or other data record, **stream-untyi** can simply decrement the pointer. If the stream does not contain a pointer into a data record, then **stream-untyi** can set a variable to the value of *char*. **stream-tyi** cooperates by checking the value of the variable; if it is non-nil, it returns the value instead of getting input from the normal source.

stream-eofp*[Stream Function]*

returns true if the stream is at its end (i.e. if there is no more data to read), and returns false if there is more data to read from the stream.

stream-eofp must be defined for all input streams.

stream-force-output*[Stream Function]*

physically writes all pending output. **stream-force-output** is used for streams which buffer their write operations. For example, it doesn't make sense to do an operating system call or a physical disk access to write a byte every time **stream-tyo** is called on a disk file. The output is stored in a buffer and written to disk after a certain accumulation or when **stream-force-output** is called. Buffering can significantly increase the speed of streams that have a high per-operation overhead (such as disk output and graphics).

stream-force-output should be defined for buffered output streams.

stream-close*[Stream Function]*

tells the stream that a program is finished with it. After being closed, the stream cannot be used for input nor for output. **stream-close** may perform various clean up operations, such as disposing of data structures which are no longer needed.

Some streams may be re-opened after they have been closed. However, when they are re-opened, they will generally have lost all of their previous state.

stream-fresh-line*[Stream Function]*

called by the Lisp function **fresh-line**. The usual version simply prints a new-line. Output streams should provide a definition of this function if they want **fresh-line** to work properly.

The general idea behind **fresh-line** is that it prints a new-line if the stream is not already at the beginning of a line.

stream-clear-input

[Stream Function]

Deletes all pending input from the stream. This function is normally only defined for buffered input streams, such as the terminal stream or serial streams. The usual version of `stream-clear-input` does nothing.

stream-listen

[Stream Function]

returns true if there are more characters to read from an input stream, nil if there are no more characters. The usual version of `stream-listen` is simply `(not (stream-eofp))`. `stream-listen` does not normally need to be redefined.

stream-abort

[Stream Function]

when the function `close` is called with a true `:abort` keyword, `stream-abort` is called. `stream-abort` should handle any bookkeeping for an abnormal closing of the stream. For example, if a file-stream is superceding an existing file, `stream-abort` (as opposed to a normal `close`) may cause the original file to be retained.

stream-column

[Stream Function]

returns the current column of the stream. This is used for tabbing purposes, and may also be used by `stream-fresh-line`.

Miscellaneous

Apropos

The search performed by `apropos` is case sensitive. This means that when `apropos` is given a string as an argument, the string should generally be all upper-case (unless you know that you are looking for a symbol containing lower-case letters). The Apropos dialog available from the Tools menu automatically converts the entered string to upper case.

Require

The function `require` maintains a list of files which are currently being loaded from calls to `require`. This list is designed to prevent circular requires from causing circular loadings.

With this system, modules can call `provide` when they are done loading, rather than when the start loading. Thus, if an error aborts the loading of a module, the module will not be improperly listed as having been provided.

Read

The functions `read` and `read-line` have been modified to behave intelligently when reading from the `*terminal-io*` (i.e., from the Listener). They now allow backspacing, cutting, pasting, etc.

Miscellaneous Functions and Variables

Two new proclamations have been added that allow variable names to always be ignored or not ignored. Executing: `(proclaim '(ignore ignore))` will inform the compiler to never generate unused lexical variable warnings for the symbol `ignore`. Executing: `(proclaim '(unignore ignore))` undoes the previous proclamation.

warn-if-redefine-kernel

[Variable]

determines whether warnings are issued when built-in non-object functions are redefined. If `*warn-if-redefine-kernel*` is `nil`, no warning is given. This variable should usually be non-`nil`, but may be bound to `nil` when redefining such functions as `short-site-name`.

`*warn-if-redefine*` is still used for user functions and built-in object functions.

compiler-warnings

[Variable]

determines whether compiler warnings are printed for incrementally compiled functions. The default is `t`.

fasl-compiler-warnings

[Variable]

the default value for the `:warnings` argument to `compile-file`. This determines whether warnings are printed when files are compiled. The default is `t`.

use-wait-next-event

[Variable]

determines whether Allegro CL should use the wait-next-event trap when idling. Using this trap gives the maximum amount of time to background processes, but it can degrade system response, especially with older versions of the Macintosh system software.

The default value is `nil`.

idle

[Variable]

signals the event system that Lisp is idle, and that it can use the wait-next-event-trap. The trap will only be used when both `*idle*` and `*wait-next-event*` are true. This variable is normally bound by the read-loop when it is awaiting input, and by modal-dialog. You can also bind it in programs which idle in other situations.

quit

[Function]

exits Allegro CL. Before exiting, all windows are closed. This is the function called by the **Quit** menu-item on the **File** menu.

uncompile-function *function-object*

[Function]

returns the uncompiled definition of a compiled function. This definition is only available if the function was compiled with `*save-definitions*` true.

function-object must be a compiled function object. It cannot be a symbol.

put-scrap
get-scrap

[Function]
[Function]

these functions transfer information between the Allegro CL scrap and the system scrap. These are the functions which allow cutting and pasting between Allegro CL, desk accessories, and other applications.

You can redefine these functions if you want to share information besides text (perhaps bitmaps, pictures, etc).

put-scrap is called whenever Allegro CL receives a suspend event, when a desk accessory window is activated, and when the application quits.

get-scrap is called whenever Allegro CL receives a resume event, when a desk accessory window is deactivated, and when the application is launched.

str-of-spaces

[Variable]

a string of 255 spaces. This string is used by the pretty-printer, and probably should not be modified.

finder-parameters

[Function]

returns a list of files which were opened with Allegro, and an indication of whether they were opened by selecting **Print** or **Open** from the Finder. The first item on the list will be the symbol :open or the symbol :print. The rest will be the files which were opened with Allegro. For example, if you start Allegro by clicking on the file "my-file", a call to finder-parameter would return the list (:open "my-file").

MultiFinder

Allegro CL now runs in the background under MultiFinder. This means that you can perform all Lisp operations—including compiling files—while using another program in the foreground.

Hardcopy Printing

Functions for calling Macintosh printer drivers have been exported. These functions correspond to the high-level functions described in Inside Macintosh (except that they have '%_' prepended to their names).

%_propen	%_prclose	%_printdefault
%_prvalidate	%_prstdialog	%_prjobdialog
%_prjobmerge	%_propendoc	%_propenpage
%_prclosepage	%_prclosedoc	%_prpicfile

In addition, the function get-print-record has been exported. This function can only be called after %_propen has been called and before %_prclose has been called. It returns a handle to print record (described in Inside Macintosh). Note: all print manager hacking must be surrounded by a call to without-interrupts. This is true because Lisp event processing interferes with the print manager. The unfortunate side-effect is that such code is difficult to debug. An example file included with version 1.2 shows how to create a class of

windows which know how to print their contents.

Changes from Version 1.1

The functions `get-selected-string` and `hardcopy-file`, documented in version 1.1, have been removed in version 1.2. Their functionality can easily be duplicated by user-defined functions.

Known Bugs

If code running from the Listener is interrupted by an event which in turn causes an error, `(continue)` goes to top level rather than resuming the listener code.

In `format` to synonym stream of standard output, `~&` causes stack overflow.

Definitions of the form `(defxxx pkg:foo ...)` or `(defxxx pkg::foo)` are not properly recognized by `edit-definition`.

When multiple kills are concatenated, the styles are not properly concatenated. This is only a problem when using EMACS style command in windows with multiple-fonts.